This is a classic "Hierarchical Configuration Management" problem. The core challenge is handling **inheritance** and **overrides** efficiently without destroying runtime performance.
Here is the design for your Health Plan Management Configuration System.

1. Design Overview

The system follows a **Multi-Level Inheritance** model. Configuration is not a flat list of settings but a tree where child nodes inherit values from parent nodes unless explicitly overridden.

- **Core Concept:** A "Configuration Object" is a JSON document containing settings (feature flags, UI toggles, business rules).
- **Inheritance:** Global Template \rightarrow Tenant \rightarrow Carrier \rightarrow Marketplace \rightarrow Market Segment \rightarrow Product.
- **Runtime Efficiency:** The backend computes the "Effective Configuration" (the final merged result). The SDK consumes this resolved state, caching it to ensure the application allows zero-latency lookups during user flows.

2. Configuration Structure & Hierarchy

We will define a fixed hierarchy for strict governance, but use flexible JSONB for the actual setting values.

The Hierarchy Levels

1. **Global/System:** Base defaults for the entire SaaS platform.
2. **Tenant:** (e.g., Cambria) High-level defaults for the organization.
3. **Carrier:** Specific settings for the insurance carrier entity within the tenant.
4. **Marketplace:** (e.g., New Jersey, Georgia) Regional overrides.
5. **Segment:** (e.g., Small Group, ICHRA) Business line overrides.
6. **Product:** (e.g., Medical, Ancillary) Product-specific module settings.

Example: "Census Module" Config

- **Level 1 (Tenant):** max_dependents: 10
- **Level 3 (Carrier - Cambria):** max_dependents: 15 (Override)
- **Level 4 (Marketplace - NJ):** requires_zip_code: true (Addition)
- **Level 5 (Segment - Small Group):** Inherits max_dependents: 15 and requires_zip_code: true.

3. Data Model (PostgreSQL Hybrid)

We will use a relational model to enforce the strict hierarchy of your entities (Tenant, Carrier, etc.) and a Document model (JSONB) for the configuration data to allow schema-less flexibility for features.

A. Entity Tables (Relational Backbone)

These tables ensure referential integrity for your business domain.

```
CREATE TABLE tenants (
```

```sql
    id UUID PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    code VARCHAR(50) UNIQUE -- e.g., 'CAMBRIA'
);

CREATE TABLE carriers (
    id UUID PRIMARY KEY,
    tenant_id UUID REFERENCES tenants(id),
    name VARCHAR(255)
);

CREATE TABLE marketplaces (
    id UUID PRIMARY KEY,
    carrier_id UUID REFERENCES carriers(id),
    state_code VARCHAR(2), -- e.g., 'NJ', 'GA'
    name VARCHAR(255)
);

CREATE TABLE market_segments (
    id UUID PRIMARY KEY,
    marketplace_id UUID REFERENCES marketplaces(id),
    segment_type VARCHAR(50) -- 'SMALL_GROUP', 'ICHRA'
);
```

B. Configuration Store (Document Model)

Instead of creating columns for every setting, we use a configurations table that links to *any* entity level.

```sql
-- Represents the raw config fragments (deltas) at a specific level
CREATE TABLE configurations (
    id UUID PRIMARY KEY,

    -- Polymorphic association to the hierarchy
    entity_type VARCHAR(50), -- 'TENANT', 'CARRIER', 'MARKETPLACE',
'TEMPLATE'
    entity_id UUID,          -- The ID of the Tenant, Carrier, etc.

    config_scope VARCHAR(50), -- 'GLOBAL', 'MODULE_CENSUS',
'MODULE_RATING'

    -- The actual settings.
    -- e.g. {"max_dependents": 15, "enable_broker_portal": true}
    data JSONB NOT NULL,

    version INT DEFAULT 1,
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP DEFAULT NOW(),
```

```
    UNIQUE(entity_type, entity_id, config_scope)
);
```

C. Templates

Templates are just configurations not attached to a live entity, used to "hydrate" a new Carrier or Marketplace.

4. Application Architecture

A. Admin UI (React + Redux/Zustand)

- **Visual Editor:** A tree-view sidebar allowing admins to select a node (e.g., "NJ Marketplace").
- **JSON Editor/Form:** A dynamic form generator based on a JSON Schema.
- **Inheritance Indicator:** When viewing "NJ Marketplace," the UI shows values in *Grey* (Inherited) and *Black* (Overridden). The user can "break inheritance" by editing a value.

B. Config Service (Java Spring Boot)

This microservice manages the definition and resolution of configs.

- **Merge Logic:** It fetches the chain (Tenant -> Carrier -> MP) and performs a "Deep Merge" of the JSONB objects.
- **Caching:** It uses Redis to cache the *Resolved Configuration* for a specific path to avoid re-computing the merge on every request.

C. Runtime SDK (Java Library)

This is critical for performance. The "Quote Management Application" (Consumer) should not make HTTP calls for every config check.

SDK Design:

1. **Initialization:** On startup (or periodically), the SDK fetches the *full resolved configuration map* for the relevant context (e.g., loaded for Cambria + NJ).
2. **Local Evaluation:** The SDK provides methods like config.getBoolean("allow_dependents"). This looks up the value in memory.
3. **Background Refresh:** The SDK uses a background thread (or via WebSockets/PubSub) to listen for config updates and invalidate the local cache without restarting the app.

5. Sequence Diagrams & CRUD Operations

Scenario 1: Admin Updates a Configuration (Write)

The Admin wants to change the max_dependents for the **NJ Marketplace**.

1. **React UI** calls GET /api/config/effective?level=MARKETPLACE&id={nj_id}.
2. **Service** calculates the merge: TenantConfig + CarrierConfig + MarketplaceConfig.
3. **Service** returns the JSON.

4. **React UI** renders the form. User changes max_dependents to 20.
5. **React UI** calls POST /api/config/override.
   - Payload: { entityType: "MARKETPLACE", entityId: "nj_id", data: { "max_dependents": 20 } }
6. **Service** saves *only the delta* to the configurations table.
7. **Service** publishes an event: CONFIG_CHANGED: MARKETPLACE:NJ.

Scenario 2: Runtime Access via SDK (Read)

The Quote Engine is calculating a rate and needs to know if "Composite Rating" is enabled.

1. **Quote Engine (Consumer)** initializes ConfigSDK with context: { carrier: "Cambria", state: "NJ" }.
2. **ConfigSDK** checks internal memory. If empty:
   - Calls **Config Service** GET /api/config/resolved/{context_hash}.
   - **Config Service** checks Redis. If miss, computes SQL Hierarchical Merge -> Stores in Redis -> Returns JSON.
3. **ConfigSDK** stores the JSON map in local heap memory.
4. **Quote Engine** calls sdk.get("rating.composite_enabled").
   - **Result:** true (Returned instantly from memory, 0ms latency).

6. Design Sequence Summary

1. **Define Schema:** Create the PostgreSQL tables (tenants, configurations JSONB).
2. **Backend Core:** Implement the ConfigMerger strategy (Spring Bean) that takes a list of JSON objects and merges them in order.
3. **API Layer:** Expose endpoints for getEffectiveConfig(context) and saveConfigDelta(context, data).
4. **SDK Implementation:** Build the Java JAR that wraps the API calls and implements a generic ConfigProvider interface with caching.
5. **Frontend:** Build the React "Inheritance Viewer" component.

7. Next Step

Would you like me to generate the **Java Code for the "Deep Merge" logic** (handling the JSONB inheritance) or the **React Component structure** for the configuration editor?